BE Ma325 : L'astuce des noyaux de type positif

Ma325, AÉRO 3, IPSA

Romain LEFOL, Simon LIGNAC, Steeve MBOCK, Florian ALAUX, Guilhem PERRET-BARDOU, Nahia ZABALA

25 Mars 2025

Table des matières

Introduction

Dans ce BE, nous allons explorer une technique couramment appelée *l'astuce des noyaux* (kernel trick). Tout au long de nos études, nous avons été amenés à utiliser l'algèbre linéaire et un large éventail de méthodes pour résoudre différents types de problèmes. Cependant, ces méthodes sont généralement limitées aux cas linéaires, bilinéaires (comme le produit scalaire) ou encore multilinéaires (comme le déterminant). Se restreindre uniquement au cadre linéaire serait regrettable. Heureusement, nous allons découvrir qu'il est possible d'étendre certains algorithmes initialement conçus pour des problèmes linéaires afin de les appliquer à des contextes non linéaires grâce à l'astuce des noyaux.

L'idée intuitive derrière cette approche est la suivante : pour résoudre un problème non linéaire, on peut chercher à représenter les données dans un espace de dimension plus élevée, où le problème devient solvable via des méthodes linéaires. Mathématiquement, cela revient à introduire une fonction de transformation :

$$\varphi : E \to \mathbb{R}^N$$

L'espace $\mathbb{R}^N$ ainsi obtenu est appelé **espace de redescription**, et la fonction φ est une **fonction de redescription**. Cependant, déterminer explicitement une telle fonction est souvent hors de portée, car cela nécessiterait une connaissance approfondie de la structure sous-jacente des données. Or, c'est précisément cette structure que l'on cherche généralement à comprendre. Heureusement, il est possible de contourner cette difficulté en exploitant certaines propriétés algébriques.

Dans ce BE, nous travaillerons dans le cadre des espaces de Hilbert de dimension finie. Pour rappel, un **espace de Hilbert** H est un espace vectoriel muni d'un produit scalaire $\langle \cdot, \cdot \rangle$ et qui est complet pour la norme induite par ce produit scalaire. Nous noterons cette norme $\| \cdot \|$ lorsqu'il n'y a pas d'ambiguïté.

Nous considérerons un ensemble E fini et non vide, ainsi qu'une fonction $\varphi : E \to H$. À partir de cette fonction, nous définissons une **fonction noyau** associée, notée k, qui s'écrit :

$$k(x, y) := \langle \varphi(x), \varphi(y) \rangle, \quad \forall (x, y) \in E \times E$$

Cette fonction noyau peut être interprétée comme une **mesure de similarité** entre les éléments x et y lorsqu'ils sont projetés dans l'espace de redescription H.

Si l'on considère une application injective $\alpha : \{1, \ldots, p\} \to E$, on note alors $x_i := \alpha(i)$ pour tout i, et on définit la **matrice de Gram associée** :

$$K(\alpha)_{ij} = k(x_i, x_j)$$

Cette matrice permet de représenter la structure des relations entre les éléments de E à travers la fonction noyau.

Chapitre 1 : Introduction et motivations

Historique de l'ACP : de sa naissance à aujourd'hui

L'analyse en composantes principales (ACP) trouve son origine au tout début du XX^{e} siècle, dans un contexte où la statistique commence à s'intéresser sérieusement à l'analyse de données multidimensionnelles. C'est **Karl Pearson** qui, en 1901, introduit pour la première fois formellement **l'idée de réduire la dimension des données tout en maximisant la conservation de l'information**. Dans son article fondateur intitulé « ***On Lines and Planes of Closest Fit to Systems of Points in Space*** », Pearson affirme que la représentation graphique et géométrique des données peut être grandement simplifiée sans altérer la structure essentielle. Selon lui, l'analyse consiste à trouver les axes de meilleure projection dans l'espace des variables, qu'il définit comme des axes de proximité maximale.

À cette époque, Pearson répond à un besoin scientifique fort : les biologistes, les anthropologues et les économistes doivent interpréter une multitude de mesures. L'ACP permet alors d'extraire des tendances globales à partir de ces nombreuses variables.

Quelques décennies plus tard, dans les années 1930, **Harold Hotelling** reprend et étend les idées de Pearson. Il introduit une **vision plus algébrique et formalise l'ACP en termes de combinaisons linéaires de variables**. Hotelling insiste sur l'idée que les nouvelles variables (les composantes principales) doivent être indépendantes les unes des autres, ce qui facilite grandement leur interprétation. Selon ses mots : « Il est souhaitable d'obtenir un ensemble de variables non corrélées, chacune représentant un aspect différent de la variabilité des données. »

Les formes que prend aujourd'hui l'ACP

Si l'ACP classique reste largement enseignée et utilisée, de nombreuses variantes adaptées aux nouveaux défis analytiques ont vu le jour.

L'ACP traditionnelle, fondée sur la matrice de covariance ou de corrélation entre variables continues, reste la méthode de référence pour l'analyse exploratoire. Toutefois, dans des contextes où les données sont qualitatives ou mixtes (par exemple, des questionnaires ou des enquêtes sociologiques), des adaptations telles que **l'Analyse Factorielle des Correspondances (AFC)** et **l'Analyse Factorielle des Correspondances Multiples (AFCM)** prolongent l'esprit de l'ACP.

Avec l'essor du **Big Data**, des méthodes rapides telles que la **Randomized PCA** permettent aujourd'hui de traiter d'importants ensembles de données sans perdre de temps ou saturer la mémoire des machines. Dans le domaine de l'intelligence artificielle, l'ACP est souvent utilisée comme technique de réduction de la dimensionnalité avant d'appliquer des algorithmes de classification ou de *clustering*. Par exemple, dans la reconnaissance faciale, l'ACP est utilisée pour extraire ce que l'on appelle les *eigenfaces*, c'est-à-dire les traits caractéristiques essentiels des visages.

Enfin, des variantes non linéaires comme le **Kernel PCA** permettent de capturer des relations complexes entre variables, en transformant les données dans des espaces de très grande dimension tout en préservant leur structure cachée.

Pour mieux comprendre, montrons quelques exemples.

Nous allons appliquer la méthode à un jeu de données comportant des iris. Chaque individu de ce jeu de données possède quatre informations : la longueur et la largeur du sépale, ainsi que la longueur et la largeur du pétale. Nous avons des vecteurs dans $\mathbb{R}^4$.

sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	0
4.9	3	1.4	0.2	0
4.7	3.2	1.3	0.2	0
4.6	3.1	1.5	0.2	0
5	3.6	1.4	0.2	0
5.4	3.9	1.7	0.4	0
4.6	3.4	1.4	0.3	0
5	3.4	1.5	0.2	0
4.4	2.9	1.4	0.2	0
4.9	3.1	1.5	0.1	0
5.4	3.7	1.5	0.2	0
4.8	3.4	1.6	0.2	0
4.8	3	1.4	0.1	0
4.3	3	1.1	0.1	0

TABLE 1 – Aperçu des premières lignes de la base de données Iris

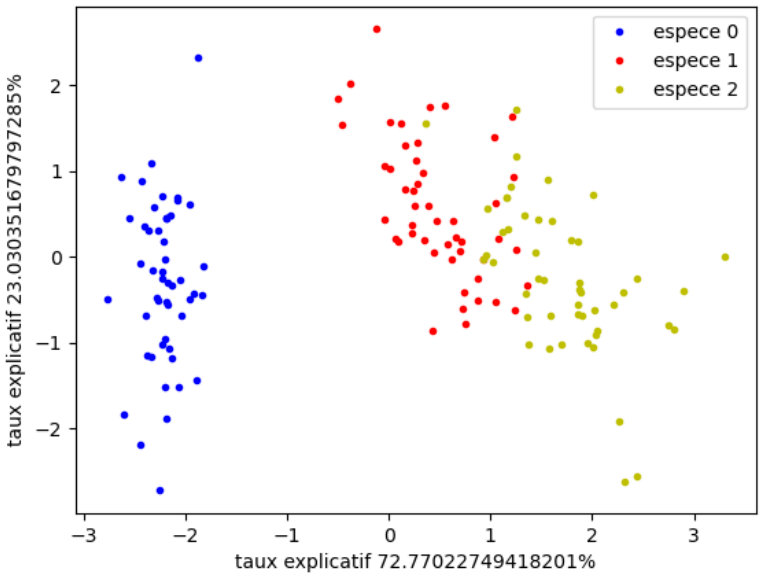


FIGURE 1 – ACP sur la base de donnée iris

Nous apercevons que l'on arrive à ramener une donnée composée de quatre composantes, donc un vecteur de dimension 4 à une projection en dimension 2. On remarque que la méthode nous donne même l'importance relative de chaque axe. Testons maintenant notre méthode sur une base de données plus complexe, « fashion minst ».

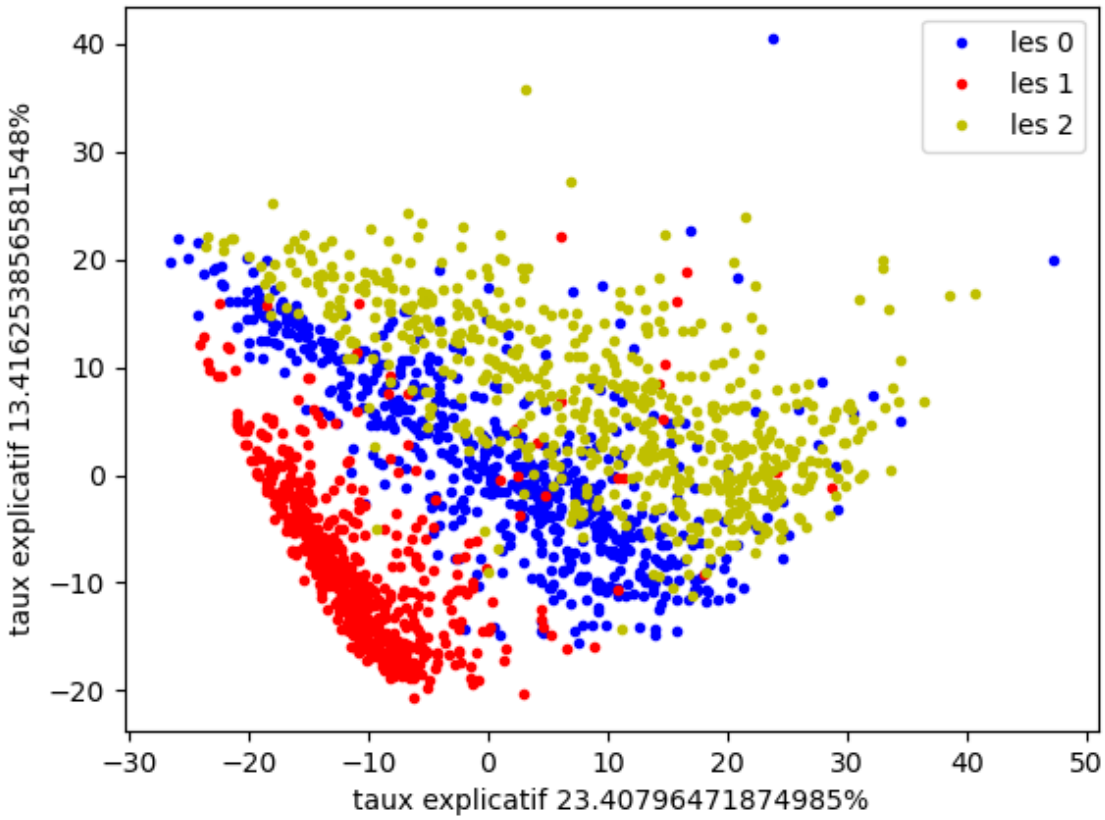


FIGURE 2 – ACP sur la base de donnée fashion mnist train

Listing 1 – Code de l'ACP

```
def ACP(data):

    m,n = np.shape(data)

    a = np.mean(data,axis=0)

    Ac = data - a

    D = np.zeros((n,n))
    print("ca roule")
    for i in range(n):

        D[i,i] = np.sqrt( np.mean(Ac[:,i]**2))

    D = D + 10**(-6)
    Acr = Ac@np.linalg.inv(D)

    U , S , VT = np.linalg.svd(Acr)
    P = np.zeros((m,n))
    taille = len(S)

    C=(1/(m*n))* VT.T @ np.diag(S**2)
    L=(1/(m*n))*S**2

    for i in range(taille):
```

```
26
27          P[:,i] = U[:,i]*S[i]
28
29      return P , L ,C
```

Ici, les choses sont plus mitigées ; il lui est plus difficile de clairement séparer les données, ce qui s'explique facilement. On dispose de vecteurs de dimension 784. Les données sont complexes, et pourtant il parvient à distinguer les 0, 1 et 2 avec seulement deux dimensions. De plus, il précise que les deux premières projections contiennent uniquement 37% des données. Peut-être faudrait-il utiliser une projection sur 3 axes pour plus de clarté ?

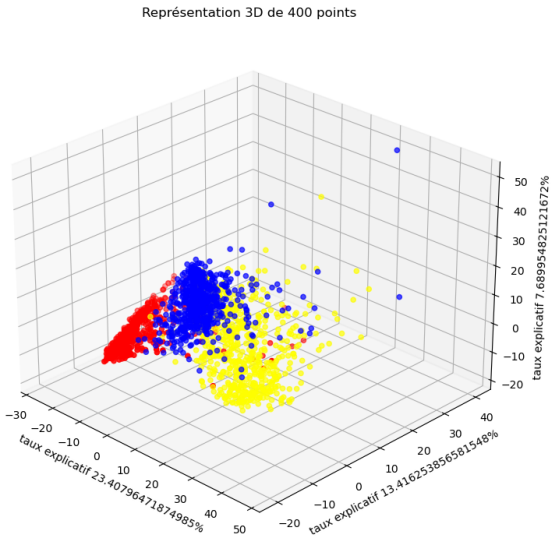


FIGURE 3 – ACP sur la base de données Fashion MNIST train — Projection sur 3 axes

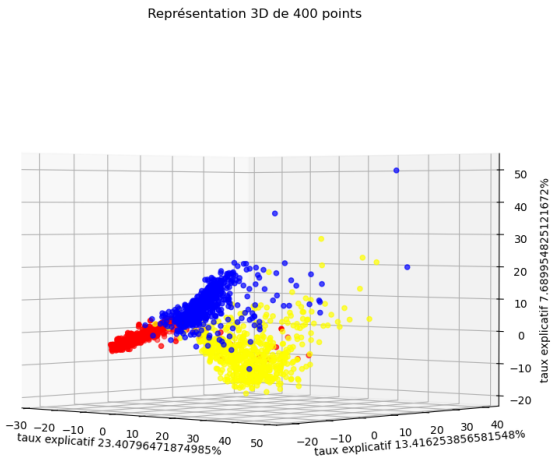


FIGURE 4 – ACP sur la base de données Fashion MNIST train — Projection sur 3 axes (vue différente)

Nous apercevons que sur trois axes, près de la moitié de l'information est contenue (43%), ce qui facilite grandement la différenciation. Nous constatons alors la force de l'ACP qui permet de simplifier l'information à traiter. Couplée à une SVM, cette méthode permet de réduire la taille de chaque donnée et donc de diminuer le coût en calcul en réduisant la taille des matrices. Ce qui accroît la vitesse.

Maintenant, les limites. Comme indiqué précédemment, l'ACP a besoin de données linéairement séparables. De plus, elle peut être sensible aux individus « spéciaux » . Voici un petit exemple. Ici, nous avons généré des points suivant deux lois normales différentes. Nous remarquons que certains points ont tendance à se chevaucher, ce qui représente nos individus « spéciaux ». Voyons comment l'ACP l'interprète. De plus la reduction de la dimmention est ega

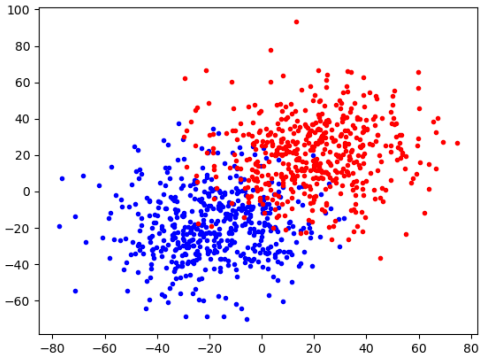


FIGURE 5 – Les points sont générés selon deux lois normales distinctes

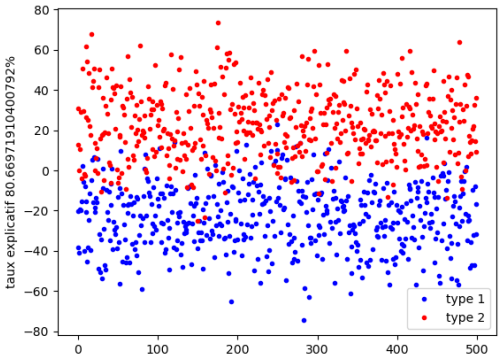


FIGURE 6 – interprétation

Nous remarquons qu'il parvient à séparer globalement nos deux lois gaussiennes, mais les points spéciaux échappent à son interprétation. Examinons ensuite la partie « les limites de la linéarité », pour laquelle le meilleur exemple visuel sont deux cercles, l'un dans l'autre, couplé à une génération aléatoire suivant une gaussienne.

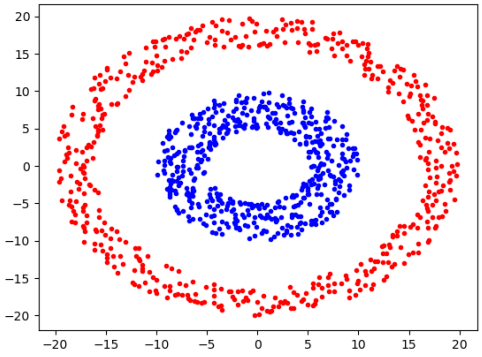


FIGURE 7 – Les points sont générés selon deux lois normales distinctes sous forme de cercle

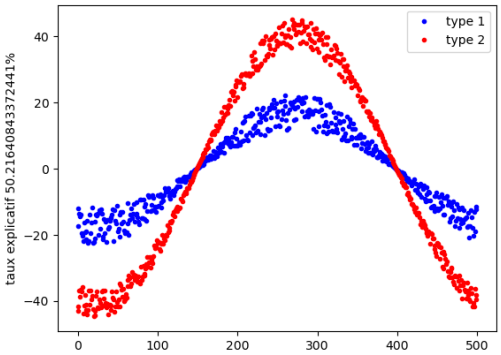


FIGURE 8 – interprétation

La limite de l'ACP est clairement atteinte ici. À l'œil nu, nous voyons clairement une séparation entre les deux groupes de points. Il n'y a pas de groupe de points spéciaux ou autre, et pourtant il ne parvient pas à les identifier. Cela est dû au fait que l'équation d'un cercle est une fonction carrée, et par définition non linéaire.

Chapitre 2 : Fondements théoriques des noyaux.

Dans ce chapitre, nous allons introduire les éléments fondamentaux de la théorie des noyaux, notamment la notion de *fonction noyau* et de *matrice de Gram*. Nous allons également démontrer une propriété essentielle : toute fonction noyau k est symétrique et sa matrice de Gram est semi-définie positive.

Propriétés des fonctions noyaux

Une fonction noyau $k : E \times E \to \mathbb{R}$ est une fonction définie à partir d'un produit scalaire dans un espace de Hilbert.

Elle vérifie la propriété suivante :

$$k(x, y) = \langle \varphi(x), \varphi(y) \rangle, \quad \forall x, y \in E.$$

Nous allons maintenant démontrer que k est une fonction symétrique et que sa matrice de Gram associée est semi-définie positive.

Symétrie de la fonction noyau

Nous voulons montrer que la fonction noyau $k(x, y)$ est symétrique, c'est-à-dire que $k(x, y) = k(y, x)$. Pour cela, partons de sa définition dans le cadre d'un espace de Hilbert.

1. **Définition du noyau**

   Le noyau $k(x, y)$ est défini à partir d'une fonction de projection φ qui associe chaque élément de notre espace d'entrée à un espace de Hilbert. Cette fonction est souvent utilisée pour transformer des données dans un espace de plus grande dimension, où elles deviennent plus faciles à séparer.

2. **Lien avec le produit scalaire**

   Par définition, la fonction noyau est donnée par :

   $$k(x, y) = \langle \varphi(x), \varphi(y) \rangle$$

   Où $\langle \cdot, \cdot \rangle$ représente le produit scalaire dans notre espace de Hilbert.

3. **Propriété du produit scalaire**
   Nous savons que dans tout espace de Hilbert, le produit scalaire possède une propriété fondamentale :

   $$\langle \varphi(x), \varphi(y) \rangle = \langle \varphi(y), \varphi(x) \rangle.$$

   Cela signifie que si nous échangeons les deux arguments de notre produit scalaire, le résultat reste le même.

Nous savons que dans tout espace de Hilbert, le produit scalaire possède une propriété fondamentale :

$$\langle \varphi(x), \varphi(y) \rangle = \langle \varphi(y), \varphi(x) \rangle.$$

Cela signifie que si nous échangeons les deux arguments de notre produit scalaire, le résultat reste le même.

4. **Conclusion**
   En appliquant cette propriété à notre noyau, nous obtenons directement :
   $$k(x,y) = \langle \varphi(x), \varphi(y) \rangle = \langle \varphi(y), \varphi(x) \rangle = k(y,x).$$
   Par conséquent, la fonction noyau est bien symétrique.

En résumé, la symétrie du noyau découle directement de la symétrie du produit scalaire dans un espace de Hilbert.

Symétrie et propriétés de la matrice de Gram

Nous allons maintenant détailler la preuve que toute fonction noyau k est symétrique et que sa matrice de Gram est semi-définie positive.

1. **Définition de la matrice de Gram**

   Considérons une famille de points $\{x_1, x_2, \ldots, x_p\} \subset E$ et la matrice de Gram associée K de taille $p \times p$ définie par :
   $$K_{ij} = k(x_i, x_j) = \langle \varphi(x_i), \varphi(x_j) \rangle.$$
   Cette matrice capture les relations entre les images des points x_i et x_j dans l'espace de Hilbert.

2. **Quadratique associée**
   Prenons un vecteur arbitraire $c = (c_1, c_2, \ldots, c_p) \in \mathbb{R}^p$. En multipliant de part et d'autre par c^T et c, nous obtenons :
   $$c^T K c = \sum_{i=1}^{p} \sum_{j=1}^{p} c_i c_j k(x_i, x_j).$$
   Cette somme représente une combinaison quadratique des termes du noyau.

3. **Interprétation via le produit scalaire**
   En utilisant la définition du noyau $k(x_i, x_j) = \langle \varphi(x_i), \varphi(x_j) \rangle$, nous pouvons réecrire cette somme sous la forme suivante :
   $$c^T K c = \sum_{i=1}^{p} \sum_{j=1}^{p} c_i c_j \langle \varphi(x_i), \varphi(x_j) \rangle.$$
   Par linéarité du produit scalaire, cette expression peut être réécrite comme :
   $$c^T K c = \left\| \sum_{i=1}^{p} c_i \varphi(x_i) \right\|^2.$$

4. **Semi-définitude positive**

   Puisque la norme d'un vecteur est toujours positive ou nulle, nous obtenons :
   $$\left\| \sum_{i=1}^{p} c_i \varphi(x_i) \right\|^2 \geq 0.$$
   Cela signifie que pour tout vecteur c, $c^T K c \geq 0$. Par définition, cela implique que la matrice de Gram K est semi-définie positive.

   Ainsi, nous avons démontré que toute fonction noyau k est symétrique et que sa matrice de Gram est semi-définie positive.

Lien entre noyaux de type positif et l'inégalité de Cauchy-Schwarz

Objectif : Dans cette partie, nous souhaitons démontrer que si $k : E \times E \to \mathbb{R}$ est un noyau de type positif, alors pour tout élément x de E, l'inégalité suivante est vérifiée : Nous avons :

$$k(x, y)^2 \leq k(x, x)k(y, y).$$

Cette inégalité est une forme généralisée de l'inégalité de Cauchy-Schwarz classique dans les espaces euclidiens. Elle joue un rôle fondamental dans la théorie des noyaux, garantissant ainsi que la fonction k se comporte comme un produit scalaire au sens généralisé.

Démonstration :

Soit $k : E \times E \to \mathbb{R}$ un noyau de type positif. Par définition, cela signifie que pour toute application injective :

$$\alpha : \{1, \ldots, p\} \to E,$$

La matrice $K(\alpha)$ définie par :

$$K(\alpha)_{ij} = k(x_i, x_j) \quad \text{où} \quad x_i = \alpha(i)$$

est **symétrique** et **semi-définie positive**.

Fixons deux éléments quelconques $x, y \in E$ et considérons l'application injective :

$$\alpha : \{1, 2\} \to E, \quad \text{avec} \quad \alpha(1) = x, \quad \alpha(2) = y.$$

La **matrice de Gram** associée est alors

$$K(\alpha) = \begin{pmatrix} k(x, x) & k(x, y) \\ k(y, x) & k(y, y) \end{pmatrix}.$$

Comme k est symétrique, on a $k(x, y) = k(y, x)$, donc $K(\alpha)$ est bien une matrice symétrique. Le fait que $K(\alpha)$ soit semi-définie positive implique que :

— Les termes diagonaux sont positifs :

$$k(x, x) \geq 0, \quad k(y, y) \geq 0,$$

— Le déterminant de $K(\alpha)$ est positif ou nul :

$$\det(K(\alpha)) = k(x, x)k(y, y) - k(x, y)^2 \geq 0.$$

Ce qui se réécrit :

$$k(x, y)^2 \leq k(x, x)k(y, y).$$

Nous avons ainsi montré que pour tout couple $(x, y) \in E^2$,

$$k(x, y)^2 \leq k(x, x)k(y, y).$$

Lien avec l'inégalité de Cauchy-Schwarz classique : L'inégalité obtenue est une généralisation naturelle de l'inégalité de Cauchy-Schwarz classique. En effet, dans un espace de Hilbert H muni d'un produit scalaire $\langle \cdot, \cdot \rangle$, on a pour tous vecteurs $u, v \in H$:

$$|\langle u, v \rangle|^2 \leq \langle u, u \rangle \langle v, v \rangle.$$

Théorème de Mercer : pour un noyau de type positif k, il existe un espace de Hilbert H et une application $\varphi : E \to H$ telle que :

$$k(x, y) = \langle \varphi(x), \varphi(y) \rangle.$$

Ainsi, l'inégalité démontrée :

$$k(x, y)^2 \leq k(x, x)k(y, y)$$

est exactement **l'inégalité de Cauchy-Schwarz** appliquée aux vecteurs $\varphi(x)$ et $\varphi(y)$ dans l'espace H.

Cette démonstration montre que les fonctions noyaux de type positif peuvent être vues comme des généralisations abstraites de notions classiques de l'algèbre linéaire, ouvrant la voie à leur utilisation dans des cadres non-linéaires complexes.

Démonstration du theorème de mercer (fini)

Démonstration. Soit $k : E \times E \to \mathbb{R}$ un noyau de type positif. Nous allons montrer qu'il existe un espace de Hilbert H muni d'un produit scalaire $\langle \cdot, \cdot \rangle$ et une application $\phi : E \to H$ tels que $k(x, y) = \langle \phi(x), \phi(y) \rangle$ pour tout $(x, y) \in E \times E$.

On définit les fonctions k_x

Pour chaque $x \in E$, définissons la fonction $k_x : E \to \mathbb{R}$ par :

$$k_x(y) = k(x, y) \quad \forall y \in E.$$

Ces fonctions seront les éléments de base pour construire notre espace.

On construit ensuite l'espace vectoriel H_k

Considérons H_k, le sous-espace vectoriel de $\mathbb{R}^E$ engendré par la famille $\{k_x\}_{x \in E}$, c'est-à-dire l'ensemble des combinaisons linéaires finies :

$$f = \sum_{i=1}^{n} a_i k_{x_i},$$

où $a_i \in \mathbb{R}$, $x_i \in E$.

Enfin, il faut définir le produit scalaire :

Définissons un produit scalaire sur H_k en posant :

$$\langle k_x, k_y \rangle = k(x, y).$$

Pour $f = \sum_{i=1}^{n} a_i k_{x_i}$ et $g = \sum_{j=1}^{m} b_j k_{y_j}$, le produit scalaire est :

$$\langle f, g \rangle = \sum_{i=1}^{n} \sum_{j=1}^{m} a_i b_j k(x_i, y_j).$$

Vérifions les propriétés :

- **Symétrie** : Puisque $k(x_i, y_j) = k(y_j, x_i)$, on a $\langle f, g \rangle = \langle g, f \rangle$.
- **Linéarité** : La définition est linéaire par construction.

— **Positivité** : Pour $f = \sum_{i=1}^{n} a_i k_{x_i}$,

$$\langle f, f \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j k(x_i, x_j) = F^T K(\alpha) F,$$

où $F = (a_1, \ldots, a_n)^T$, $K(\alpha)_{ij} = k(x_i, x_j)$. Comme k est de type positif, K est semi-définie positive, donc $a^T K a \geq 0$.

Pour que H_k soit un espace de Hilbert, il doit être complet pour la norme $\|f\| = \sqrt{\langle f, f \rangle}$. Nous prenons la complétion de l'espace engendré par $\{k_x\}_{x \in E}$, obtenant ainsi un espace de Hilbert H_k. Le produit scalaire s'étend à H_k.

Pour terminer On définis $\phi : E \to H_k$ par :

$$\phi(x) = k_x.$$

Alors :

$$\langle \phi(x), \phi(y) \rangle = \langle k_x, k_y \rangle = k(x, y).$$

Conclusion. Nous avons construit un espace de Hilbert H_k et une application $\phi : E \to H_k$ tels que $k(x, y) = \langle \phi(x), \phi(y) \rangle$. Cela démontre bien le théorème. □

Propriétés générales sur les noyaux

Démonstrations que les fonctions suivantes sont des noyaux de type positif

Soient k_1 et k_2 deux fonctions noyaux de type positif sur $E \times E$. Définissons $k_{\text{som}}(x, y) := k_1(x, y) + k_2(x, y)$. Pour toute application injective $\alpha : \{1, \ldots, p\} \to E$, la matrice de Gram associée K_{som} est la somme des matrices de Gram K_1 et K_2 associées à k_1 et k_2 :

$$K_{\text{som}} = K_1 + K_2$$

Or, la somme de deux matrices semi-définies positives est encore semi-définie positive. Ainsi, k_{som} est une fonction noyau de type positif.
Définissons $k_{\text{prod}}(x, y) := k_1(x, y) \cdot k_2(x, y)$.

> On utilise les propriétes du produit d'Hadamard : le produit terme à terme de deux matrices semi-définies positives est encore semi-définie positive.

Comme K_1 et K_2 sont semi-définies positives, alors $K_{\text{prod}} := K_1 \circ K_2$ l'est aussi. Donc k_{prod} est aussi une fonction noyau de type positif.

Autres fonctions noyau de type positives

Soit k_1 une fonction noyau de type positif.
Pour tout polynôme P à coefficients positifs : $k(x, y) := P(k_1(x, y))$ est une fonction noyau de type positif.
En effet, selon le théorème spectral, les fonctions polynomiales à coefficients positifs appliquées à une matrice semi-définie positive préservent cette propriété.
La fonction $k(x, y) := \exp(k_1(x, y))$ est également une fonction noyau de type positif.
Cela vient du fait que l'exponentielle est une somme infinie de polynômes à coefficients positifs :

$$\exp(k_1(x, y)) = \sum_{n=0}^{\infty} \frac{k_1(x, y)^n}{n!}$$

Donc k est limite de noyaux de type positif, et cette limite est aussi un noyau de type positif.

Distance dans l'espace de redescription

Soit k un noyau de type positif, et $\varphi : E \to \mathcal{H}$ une fonction de redescription telle que $k(x, y) = \langle \varphi(x), \varphi(y) \rangle$. Alors la distance dans l'espace de Hilbert $\mathcal{H}$ est donnée par :

$$\begin{aligned} \|\varphi(x) - \varphi(y)\|_{\mathcal{H}}^2 &= \langle \varphi(x) - \varphi(y), \varphi(x) - \varphi(y) \rangle \\ &= \langle \varphi(x), \varphi(x) \rangle + \langle \varphi(y), \varphi(y) \rangle - 2\langle \varphi(x), \varphi(y) \rangle \\ &= k(x, x) + k(y, y) - 2k(x, y) \end{aligned}$$

Application numérique avec Python Nous pouvons fixer un point x_0 et calculer $\|\varphi(x_0) - \varphi(y)\|^2$ pour y dans $[-2, 2]^2$, avec différents noyaux :

Noyau gaussien :

$$k(x, y) := \exp\left(-\frac{\|x - y\|^2}{2\sigma^2}\right)$$

Noyau exponentiel :

$$k(x, y) := \exp(\langle x, y \rangle)$$

Noyau polynomial :

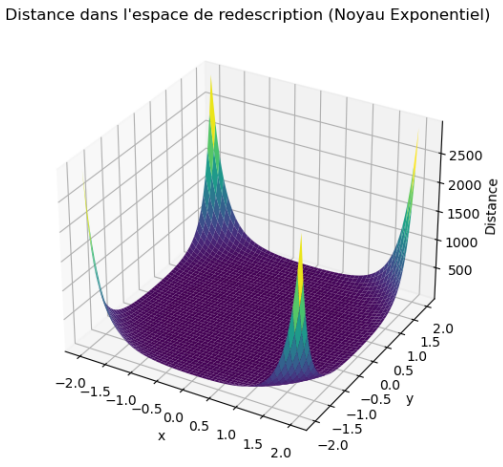
$$k(x, y) := (\langle x, y \rangle + c)^d$$

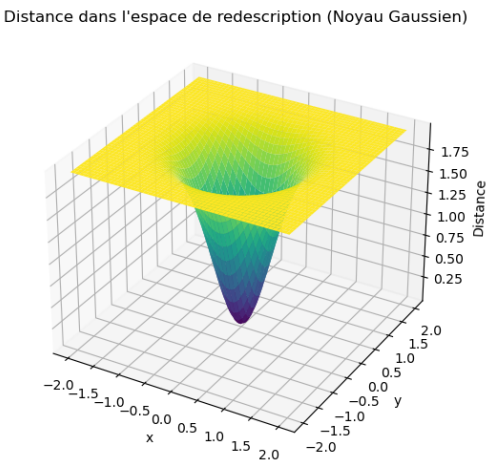
En implémentant ces noyaux dans Python et en calculant la distance cela donne :

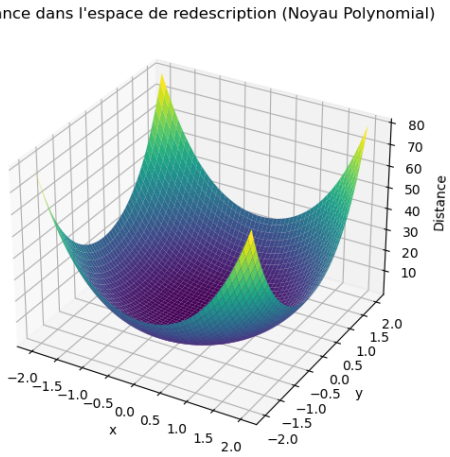
```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm


def noyau_gaussien(x, y, sigma=1.0):
    return np.exp(-np.linalg.norm(x - y)**2 / (2 * sigma**2))

def noyau_exponentiel(x, y):
    return np.exp(np.dot(x, y))

def noyau_polynomial(x, y, c=1, d=2):
    return (np.dot(x, y) + c)**d

def distance_noyau(k, x0, y):
    return k(x0, x0) + k(y, y) - 2 * k(x0, y)

x0 = np.array([0.0, 0.0])
grid = np.linspace(-2, 2, 200)
X, Y = np.meshgrid(grid, grid)


def visualiser_distance(nom, noyau):
    Z = np.zeros_like(X)
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            y = np.array([X[i, j], Y[i, j]])
            Z[i, j] = distance_noyau(noyau, x0, y)

    fig = plt.figure(figsize=(6, 5))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X, Y, Z, cmap=cm.viridis)
    ax.set_title(f"Distance dans l'espace de redescription ({nom})")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("Distance")
    plt.tight_layout()
    plt.show()


visualiser_distance("Noyau Gaussien", lambda x, y: noyau_gaussien(x, y, sigma=0.5))
```

```
42 visualiser_distance("Noyau Exponentiel", noyau_exponentiel)
43 visualiser_distance("Noyau Polynomial", lambda x, y: noyau_polynomial(x, y, c=1, d=2))
```

Ce programme Python nous permet d'avoir les graphes suivants, représentant les distances pour chaque noyau dans l'espace de redescription pour $[-2, 2]^2$


Distance dans l'espace de redescription (Noyau Exponentiel)
Distance
x
y


Distance dans l'espace de redescription (Noyau Gaussien)
Distance
x
y


Distance dans l'espace de redescription (Noyau Polynomial)
Distance
x
y

Chapitre 3 : Généralisation des moindres carrés par l'astuce des noyaux.

La méthode des moindres carrés est un outil efficace afin d'ajuster un modèle linéaire aux données. Cependant ce modèle atteint ses limites lorsqu'il en vient a traiter des structures non linéaires. C'est dans ce contexte que nous introduirons une extension de la méthodes de moindres carrés nommé l'astuce des noyaux. Cette astuce va nous permettre de projeter implicitement les données dans un espace de Hilbert de dimension supérieur à l'aide d'une fonction noyau $k(x, x')$. Cette extension est donc intéressantes car elle adopte la non linéarité du modèle tout en conservant la simplicité de la méthodes des moindres carrés.

Généralisation des moindres carrés par la théorie des noyaux

Le problème classique des moindres carrés est le suivant :

$$x^* = \arg\min_{x\in\mathbb{R}^n} \|Ax - b\|_2^2$$

où $A \in \mathcal{M}_{m,n}(\mathbb{R})$ et $b \in \mathbb{R}^m$.

Reformulation du problème

Posons $W = A^T$. Notons w_i le i-ème vecteur colonne de W, et b_i la i-ème composante de b. Nous développe :

$$\|Ax - b\|_2^2 = (Ax - b)^T(Ax - b) = \sum_{i=1}^{m}((Ax)_i - b_i)^2$$

Or, $(Ax)_i$ est la i-ème composante de Ax, qui est égale au produit scalaire de la i-ème ligne de A avec x. La i-ème ligne de A est w_i^T, donc :

$$\|Ax - b\| = \|(A^T)^T x - b\|$$

d'où :

$$\|W^T x - b\|_2^2$$

$$\sum(W_i^T x - b_i)^2$$

Ce qui donne la nouvelle formulation du problème :

$$x^* = \arg\min_{x\in\mathbb{R}^n} \sum_{i=1}^{m}(w_i^T x - b_i)^2$$

Passage aux noyaux positifs

Supposons donné un noyau $k : E \times E \to \mathbb{R}$.

Par définition, un noyau positif k garantit l'existence d'un espace de Hilbert $\mathcal{H}$ et d'une application de redescription $\varphi : E \to \mathcal{H}$ telle que :

$$k(x, y) = \langle\varphi(x), \varphi(y)\rangle_{\mathcal{H}}$$

où $\langle\cdot,\cdot\rangle_{\mathcal{H}}$ est le produit scalaire dans $\mathcal{H}$. Ainsi, nous pouvons remplacer chaque produit scalaire $w_i^T x$ par :

$$\langle x, \varphi(w_i)\rangle$$

et réécrire le problème sous la forme :

$$x^* = \arg\min_{x\in\mathcal{H}} \sum_{i=1}^{m}\left(\langle x, \varphi(w_i)\rangle - b_i\right)^2$$

Par le théorème de la représentation de Riesz, la solution optimale x^* appartient au sous-espace engendré par les $\varphi(w_i)$, soit :

$$x = \sum_{i=1}^{m} a_i \varphi(w_i)$$

avec $a = (a_1, a_2, \ldots, a_m)^T \in \mathbb{R}^m$. En injectant cette forme dans le problème :

$$\langle x, \varphi(w_i) \rangle = \left\langle \sum_{j=1}^{m} a_j \varphi(w_j), \varphi(w_i) \right\rangle = \sum_{j=1}^{m} a_j \langle \varphi(w_j), \varphi(w_i) \rangle = \sum_{j=1}^{m} a_j k(w_j, w_i)$$

Définissons la matrice de Gram $K \in \mathbb{R}^{m \times m}$ par :

$$K_{ij} = k(w_i, w_j)$$

Alors :

$$(Ka)_i = \sum_{j=1}^{m} K_{ij} a_j$$

Le problème se reformule alors :

$$a^* = \arg\min_{a \in \mathbb{R}^m} \|Ka - b\|_2^2$$

Application à l'interpolation simple

Nous allons considérer un exemple simple ici.

Méthode des carrés et droite afine

Supposons que nous cherchons sous forme d'une droite affine :

$$y = \alpha t + \beta$$

avec t, α, β réels, les coefficients α et β tels que Δ passe au plus près des points :

$$\{(-1, 1), (0, 0), (1, 1)\}$$

Nous allons montrer que ce problème se résout à l'aide de la méthode des carrés. Tout d'abord la méthode des moindres carrés consiste à déterminer les coefficients α et β qui minimisent la somme des écarts quadratiques entre nos trois points. Nous pouvons écrire :

- $(t_1, y_1) = (-1, 1)$,
- $(t_2, y_2) = (0, 0)$,
- $(t_3, y_3) = (1, 1)$.

et la fonction S qui représente la somme des écarts quadratiques à minimiser est :

$$S(\alpha, \beta) = \sum_{i=1}^{3} (y_i - (\alpha t_i + \beta))^2$$

En développant avec les points donnés nous avons :

$$S(\alpha, \beta) = (1 - (\alpha(-1) + \beta))^2 + (0 - (\alpha(0) + \beta))^2 + (1 - (\alpha(1) + \beta))^2$$

Soit :

$$S(\alpha, \beta) = (1 + \alpha - \beta)^2 + \beta^2 + (1 - \alpha - \beta)^2$$

Afin de minimiser S, résolvons les dérivées partielles égales à zéro :

$$\begin{cases} \dfrac{\partial S}{\partial \alpha} = 2(1+\alpha-\beta) - 2(1-\alpha-\beta) = 4\alpha = 0 \\ \dfrac{\partial S}{\partial \beta} = -2(1+\alpha-\beta) + 2\beta - 2(1-\alpha-\beta) = -4 + 6\beta = 0 \end{cases}$$

Nous trouvons alors :

$$\alpha = 0, \quad \beta = \frac{2}{3}$$

Ainsi, la droite affine obtenue est :

$$y = \alpha t + \beta = 0 \cdot t + \frac{2}{3}$$

$$\boxed{y = \frac{2}{3}}$$

Nous pouvons le vérifier par :

$$S\left(0, \frac{2}{3}\right) = \left(1 - \frac{2}{3}\right)^2 + \left(0 - \frac{2}{3}\right)^2 + \left(1 - \frac{2}{3}\right)^2 = \frac{1}{9} + \frac{4}{9} + \frac{1}{9} = \frac{2}{3}$$

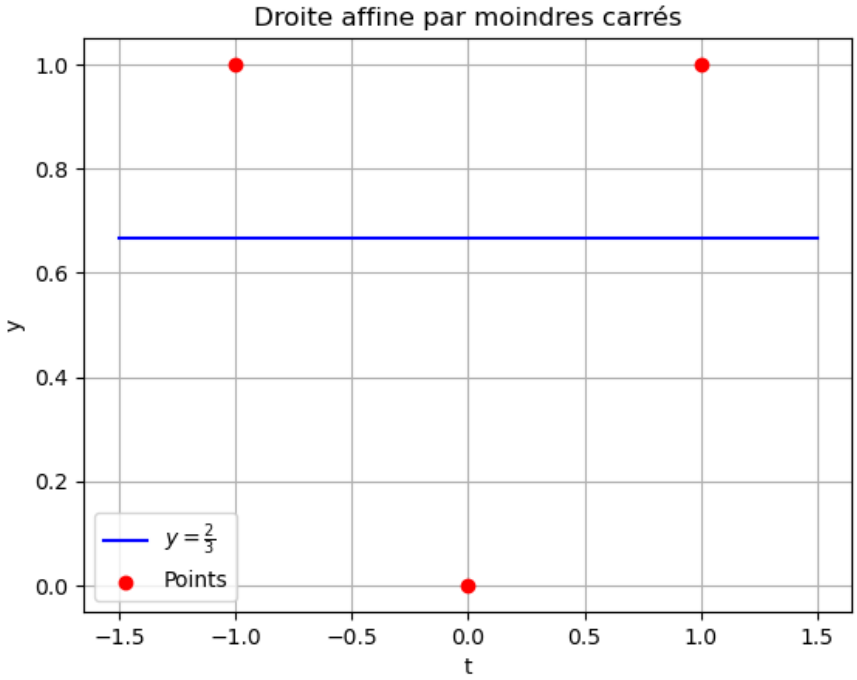


FIGURE 10 – Droite affine par moindres carrés

Astuce des noyaux et parabole

Après avoir résolue ce problème d'interpolation avec la méthode des carré. Nous allons désormais utiliser l'astuce des noyaux qui va projetter les données dans un espace caractéristiques de dimension plus élevée . Dans notre études, nous utiliserons un noyaux polynomial de degré 2 défini par :

$$K(t, s) = (ts + 1)^2$$

Nous devons désormais déterminer la matrice de Gram K :

La matrice de Gram K, construite à partir du noyau polynomial ci-dessus est une matrice carrée dont ses éléments sont définient par $K_{ij} = K(t_i, t_j)$, rappellons :

— $(t_1, y_1) = (-1, 1)$,
— $(t_2, y_2) = (0, 0)$,
— $(t_3, y_3) = (1, 1)$.

Nous noterons t_i les points d'entraînement et y_i leurs valeurs cibles associées, avec $i \in \{1, 2, 3\}$, correspondant aux points $\{(-1, 1), (0, 0), (1, 1)\}$.

Appliquons la formule :

$$\begin{aligned} K(-1,-1) &= ((-1)\cdot(-1)+1)^2 = (1+1)^2 = 4, \\ K(0,0) &= (0\cdot 0+1)^2 = (0+1)^2 = 1, \\ K(1,1) &= (1\cdot 1+1)^2 = (1+1)^2 = 4, \\ K(-1,0) &= ((-1)\cdot 0+1)^2 = (0+1)^2 = 1, \\ K(-1,1) &= ((-1)\cdot 1+1)^2 = (-1+1)^2 = 0, \\ K(0,1) &= (0\cdot 1+1)^2 = (0+1)^2 = 1. \end{aligned}$$

De plus, la matrice de Gram est symétrique, en effet : $K(t, s) = K(s, t)$, par conséquent nous avons :

$$K = \begin{bmatrix} 4 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 4 \end{bmatrix}$$

Notons y le vecteur associé aux valeurs cibles des points d'entraînement :

$$y = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

et résolvons le système $Ka = y$:

$$\begin{bmatrix} 4 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 4 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Soit,

$$\begin{cases} 4a_1 + a_2 = 1 \\ a_1 + a_2 + a_3 = 0 \\ a_2 + 4a_3 = 1 \end{cases}$$

Nous trouvons donc :

$$a = \begin{bmatrix} \frac{1}{2} \\ -1 \\ \frac{1}{2} \end{bmatrix}$$

Après avoir calculé la matrice de Gram K et résolu le système $Ka = y$, nous avons obtenu les coefficients :

$$a = \begin{bmatrix} \frac{1}{2} \\ -1 \\ \frac{1}{2} \end{bmatrix}.$$

La fonction solution dans le cadre des moindres carrés à noyau est donnée par :

$$f_K(t) = \sum_{i=1}^{3} a_i K(t_i, t),$$

où $K(t_i, t) = (t_i t + 1)^2$.

En développant la somme nous obtenons :

$$f_K(t) = \frac{1}{2}(t^2 - 2t + 1) + (-1) \cdot 1 + \frac{1}{2}(t^2 + 2t + 1).$$

En simplifiant nous retrouvons la parabole qui représente l'interpolation des points donnés :

$$\boxed{f_K(t) = t^2.}$$

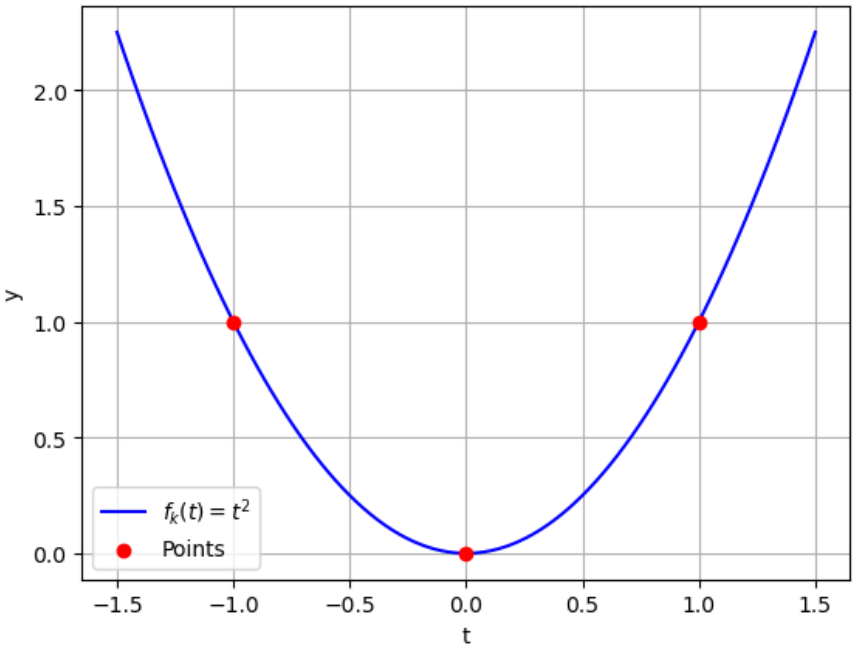


FIGURE 11 – Courbe de $f_k(t) = t^2$

On peut justifier cette forme car la régression dans l'espace des noyaux consiste à exprimer la solution comme une combinaison linéaire des noyaux, où les coefficients sont déterminés par la solution des moindres carrés.

Version multi-classe des moindres carrés à noyaux

Grâce à python on peut coder une version multi-classe des moindres carrés à noyaux. On testera notre programme avec les données MNIST présent sur moodle. :

Listing 2 – Test modèle sur données MNIST (fonctions)

```python
import matplotlib.pyplot as plt
import cvxpy as cp
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score

# Fonction pour construire la matrice de Gram a partir d'un noyau
def compute_kernel_matrix(X, kernel_func):
    m = X.shape[0]
    K = np.zeros((m, m))
    for i in range(m):
        for j in range(m):
            K[i, j] = kernel_func(X[i], X[j])
    return K

# Exemple de noyau gaussien (RBF)
def gaussian_kernel(x, y, sigma=1.0):
    return np.exp(-np.linalg.norm(x - y)**2 / (2 * sigma**2))

# Algorithme des moindres carres a noyaux multiclasse
class MCNMulticlass:
    def __init__(self, kernel_func, lambda_reg=1e-4):
        self.kernel_func = kernel_func
        self.lambda_reg = lambda_reg
        self.X_train = None
        self.A = None

    def fit(self, X, y):
        """
        X : matrice des echantillons d'entrainement (m x n)
        y : vecteur des labels (m), valeurs de 0 a c-1
        """
        self.X_train = X
        m = X.shape[0]
        c = len(np.unique(y))  # Nombre de classes

        # Construction de la matrice de labels Y (m x c)
        Y = np.zeros((m, c))
        for i in range(m):
            Y[i, y[i]] = 1

        # Calcul de la matrice de Gram K
        K = compute_kernel_matrix(X, self.kernel_func)

        # Ajout de la regularisation et resolution pour A
        I = np.eye(m)
        self.A = np.linalg.solve(K + self.lambda_reg * I, Y)

    def predict(self, X_test):
        """
        X_test : matrice des echantillons de test (m_test x n)
        Retourne les predictions des classes
        """
        m_test = X_test.shape[0]
        m_train = self.X_train.shape[0]

        # Calcul du vecteur de noyau pour chaque echantillon de test
        K_test = np.zeros((m_test, m_train))
        for i in range(m_test):
            for j in range(m_train):
                K_test[i, j] = self.kernel_func(X_test[i], self.X_train[j])

        # Prediction
        Y_pred = K_test @ self.A
        return np.argmax(Y_pred, axis=1)
```

Listing 3 – Test modèle sur données MNIST (application

```
# Test avec donnees MNIST
# Charger les donnees MNIST depuis les fichiers CSV sans en-tete
train_data = pd.read_csv('Data/mnist_train.csv', header=None)
test_data = pd.read_csv('Data/mnist_test.csv', header=None)

# Extraction des labels (premiere colonne) et des features (colonnes restantes)
y_train = train_data.iloc[:, 0].values  # Colonne 0 pour les labels
X_train = train_data.iloc[:, 1:].values  # Colonnes 1 a 784 pour les pixels
y_test = test_data.iloc[:, 0].values
X_test = test_data.iloc[:, 1:].values

# Normalisation des pixels (valeurs de 0 a 255 -> 0 a 1)
X_train = X_train / 255.0
X_test = X_test / 255.0

# Limiter le nombre d'echantillons pour accelerer les calculs
max_train_samples = 1000
max_test_samples = 500
X_train = X_train[:max_train_samples]
y_train = y_train[:max_train_samples]
X_test = X_test[:max_test_samples]
y_test = y_test[:max_test_samples]

# Initialisation du modele avec noyau gaussien
sigma = 10.0  # Parametre du noyau RBF
lambda_reg = 0.01  # Parametre de regularisation
model = MCNMulticlass(kernel_func=lambda x, y: gaussian_kernel(x, y, sigma=sigma),
                      lambda_reg=lambda_reg)

# Entrainement du modele
print("Entrainement du modele...")
model.fit(X_train, y_train)

# Prediction sur les donnees de test
print("Prediction sur les donnees de test...")
y_pred = model.predict(X_test)

# Calcul de la precision
accuracy = accuracy_score(y_test, y_pred)
print(f"Precision sur l'ensemble de test : {accuracy * 100:.2f}%")
```

Avec cette base de donnée on obtient une précision sur l'ensemble de test de 90.60%

Généralisation des problèmes de moindres carrés avec régularisation au cas à noyaux

Dans le problème des moindres carrés classique avec régularisation L2 (Ridge), on cherche à minimiser :

$$x^* = \arg\min_{x \in \mathbb{R}^n} \|Ax - b\|_2^2 + \lambda \|x\|_2^2,$$

où $A \in \mathcal{M}_{m,n}(\mathbb{R})$, $b \in \mathbb{R}^m$, et $\lambda > 0$ est le paramètre de régularisation. La solution est donnée par :

$$x^* = (A^\top A + \lambda I)^{-1} A^\top b.$$

Dans le cadre des noyaux, on suppose que les données w_i (colonnes de $A^\top$) sont transformées via une fonction de redescription $\varphi : E \to \mathcal{H}$, où $\mathcal{H}$ est un espace de Hilbert. Le problème devient :

$$x^* = \arg\min_{x \in \mathcal{H}} \sum_{i=1}^{m} \left(\langle x, \varphi(w_i) \rangle - b_i \right)^2 + \lambda \|x\|_{\mathcal{H}}^2,$$

où $\| \cdot \|_{\mathcal{H}}$ est la norme dans $\mathcal{H}$.

Selon le théorème de représentation, la solution x^* peut être exprimée comme :

$$x^* = \sum_{i=1}^{m} a_i \varphi(w_i),$$

où $a = (a_1, \ldots, a_m)^\top \in \mathbb{R}^m$.

Substituons dans l'objectif :

$$\langle x, \varphi(w_i) \rangle = \sum_{j=1}^{m} a_j \langle \varphi(w_j), \varphi(w_i) \rangle = \sum_{j=1}^{m} a_j k(w_j, w_i) = (Ka)_i,$$

où K est la matrice de Gram avec $K_{ij} = k(w_i, w_j)$.

La norme de x dans $\mathcal{H}$ est :

$$\|x\|_{\mathcal{H}}^2 = \left\langle \sum_{i=1}^{m} a_i \varphi(w_i), \sum_{j=1}^{m} a_j \varphi(w_j) \right\rangle = \sum_{i,j=1}^{m} a_i a_j k(w_i, w_j) = a^\top K a.$$

Ainsi, le problème se réécrit :

$$a^* = \arg\min_{a \in \mathbb{R}^m} \|Ka - b\|_2^2 + \lambda a^\top K a.$$

Pour trouver a^*, on calcule le gradient de l'objectif par rapport à a :

$$\nabla_a \left(\|Ka - b\|_2^2 + \lambda a^\top K a \right) = 2K^\top (Ka - b) + 2\lambda K a.$$

En posant le gradient à zéro, on obtient :

$$K^\top K a - K^\top b + \lambda K a = 0 \quad \Rightarrow \quad (K^\top K + \lambda K) a = K^\top b.$$

Si K est inversible (ce qui est souvent le cas avec une régularisation adéquate ou un noyau bien choisi), on peut résoudre :

$$a = (K + \lambda I)^{-1} b,$$

car $K^\top = K$ (la matrice de Gram est symétrique).

Cette solution correspond à la généralisation de la régression Ridge avec noyaux.

Pour la régularisation L1 (Lasso), le problème devient :

$$a^* = \arg\min_{a \in \mathbb{R}^m} \|Ka - b\|_2^2 + \lambda \|a\|_1.$$

Contrairement à la régularisation L2, ce problème n'a pas de solution analytique simple en raison de la non-différentiabilité de la norme L1.

La prédiction pour un nouveau point w est donnée par :

$$f(w) = \langle x^*, \varphi(w) \rangle = \sum_{i=1}^{m} a_i k(w_i, w).$$

Listing 4 – Moindres carres avec regularisation L2 a noyaux (fonctions

```
def gaussian_kernel(x, y, sigma=1.0):
    """Noyau gaussien (RBF)."""
    return np.exp(-np.linalg.norm(x - y)**2 / (2 * sigma**2))

def compute_kernel_matrix(X, kernel_func):
    """Calcule la matrice de Gram pour un noyau donne."""
    m = X.shape[0]
    K = np.zeros((m, m))
```

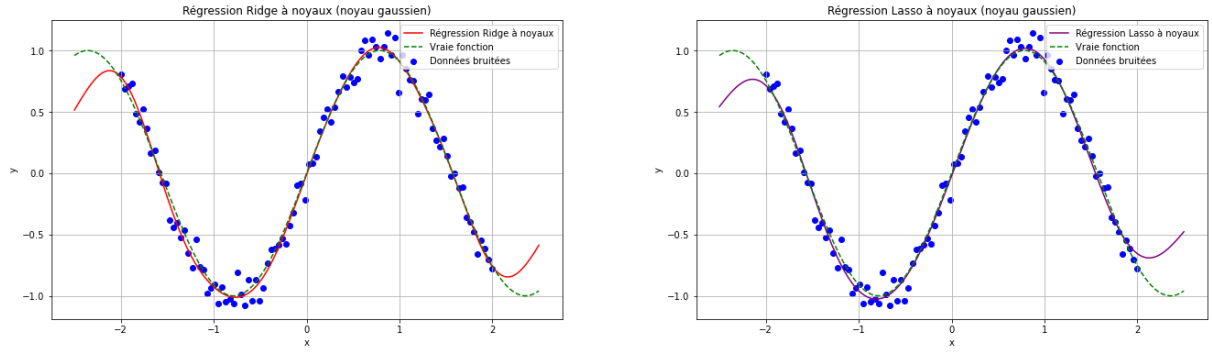
```
    for i in range(m):
        for j in range(m):
            K[i, j] = kernel_func(X[i], X[j])
    return K

def kernel_ridge_regression(X, y, kernel_func, lambda_reg=1e-4):

    K = compute_kernel_matrix(X, kernel_func)
    m = X.shape[0]
    I = np.eye(m)
    a = np.linalg.solve(K + lambda_reg * I, y)
    return a, X

def predict_kernel_ridge(X_test, X_train, a, kernel_func):

    m_test = X_test.shape[0]
    m_train = X_train.shape[0]
    K_test = np.zeros((m_test, m_train))
    for i in range(m_test):
        for j in range(m_train):
            K_test[i, j] = kernel_func(X_test[i], X_train[j])
    y_pred = K_test @ a
    return y_pred

def kernel_lasso_regression(X, y, kernel_func, lambda_reg=1e-4):

    K = compute_kernel_matrix(X, kernel_func)
    m = X.shape[0]
    a = cp.Variable(m)
    objective = cp.Minimize(0.5 * cp.sum_squares(K @ a - y) + lambda_reg * cp.norm1(a))
    problem = cp.Problem(objective)
    problem.solve()
    return a.value, X
```

Listing 5 – Moindres carres avec regularisation L2 a noyaux (application)

```

# Generation de donnees synthetiques non lineaires
np.random.seed(42)
n_samples = 100
X = np.linspace(-2, 2, n_samples).reshape(-1, 1)
y = np.sin(2 * X).ravel() + 0.1 * np.random.randn(n_samples)  # Signal bruite

# Parametres
sigma = 0.5  # Parametre du noyau gaussien
lambda_reg = 0.01  # Parametre de regularisation

# Entrainement du modele
a, X_train = kernel_ridge_regression(X, y, kernel_func=lambda x,
                        y: gaussian_kernel(x, y, sigma=sigma),
                                    lambda_reg=lambda_reg)

# Prediction sur une grille pour visualisation
X_test = np.linspace(-2.5, 2.5, 200).reshape(-1, 1)
y_pred = predict_kernel_ridge(X_test, X_train,
                    a, lambda x,
                    y: gaussian_kernel(x, y, sigma=sigma))

# Visualisation
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', label='Donnees bruittees')
plt.plot(X_test, y_pred, color='red', label='Regression Ridge a noyaux')
plt.plot(X_test, np.sin(2 * X_test), color='green', linestyle='--', label='Vraie fonction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Regression Ridge a noyaux (noyau gaussien)')
plt.legend()
plt.grid(True)
```

```
plt.show()

# Analyse des resultats
print("Erreur quadratique moyenne sur les donnees d'entrainement :",
      np.mean((predict_kernel_ridge(X, X_train,
                a, lambda x,
                y: gaussian_kernel(x, y, sigma=sigma)) - y)**2)
                )

# Chapitre 3d : Moindres carres avec regularisation L1 a noyaux

# Entrainement du modele Lasso
a_lasso, X_train = kernel_lasso_regression(X, y, lambda x,
                                    y: gaussian_kernel(x, y, sigma=sigma),
                                    lambda_reg=0.1)

# Prediction
y_pred_lasso = predict_kernel_ridge(X_test, X_train,
            a_lasso, lambda x,
            y: gaussian_kernel(x, y, sigma=sigma)
            )

# Visualisation
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', label='Donnees bruittees')
plt.plot(X_test, y_pred_lasso, color='purple', label='Regression Lasso a noyaux')
plt.plot(X_test, np.sin(2 * X_test), color='green', linestyle='--', label='Vraie fonction')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Regression Lasso a noyaux (noyau gaussien)')
plt.legend()
plt.grid(True)
plt.show()

# Analyse
print("Erreur quadratique moyenne (Lasso) :",
      np.mean((predict_kernel_ridge(X, X_train,
                a_lasso, lambda x,
                y: gaussian_kernel(x, y, sigma=sigma)) - y)**2)
                )
```

Avec ce code on obtient les deux résultats suivants : Erreur quadratique moyenne sur les données d'entraînement : 0.007213651007535011 Erreur quadratique moyenne (Lasso) : 0.008244693563194598



(a) Resultats Ridge

(b) Resultats Lasso

FIGURE 12 – Deux images cote a cote

Chapitre 4 : Analyse en Composantes Principales (ACP) à noyaux

Ce chapitre s'inscrit dans la continuité du chapitre 5 sur l'ACP classique. Nous nous concentrerons en particulier sur la généralisation par les noyaux, en mettant l'accent sur les aspects pratiques mais aussi les limites de cette approche. Certaines démonstrations sont volontairement simplifiées pour privilégier l'intuition géométrique.

L'Analyse en Composantes Principales classique repose sur une décomposition spectrale de la matrice de covariance des données. Cette approche linéaire permet d'identifier les directions de plus grande variance dans les données. Cependant, cette linéarité limite son applicabilité à des structures plus complexes.

Considérons par exemple deux nuages de points en forme de :

— Cercles concentriques dans $\mathbb{R}^2$

— Spirale logarithmique

— Spheres impriquées dans $\mathbb{R}^3$

Dans ces cas, l'ACP classique échoue à révéler la structure sous-jacente du fait qu'elle ne peut capturer que les relations linéaires entre les variables. L'astuce du noyau permet d'éviter cette limitation en projetant implicitement les données dans un espace de dimension supérieure.

Fondements théoriques

Rappel d'ACP classique

Soit une matrice de données $R \in \mathbb{M}_{m,n}(\mathbb{R})$, où chaque ligne représente un individu et chaque colonne une variable.
Nous procédons en deux étapes :

1. **Centrage** : nous soustrayons le vecteur moyen $x^\bullet = \frac{1}{m}\sum_{i=1}^{m} x_{i\bullet}$ à chaque ligne :

$$R^{(c)} = R - \mathbf{1}_m x^\bullet$$

2. **Changement de repère** : nous cherchons une base orthonormée maximisant la variance projetée. Cette base est obtenue en diagonalisant la matrice de covariance :

$$C = \frac{1}{m}(R^{(c)})^T R^{(c)}$$

ou équivalemment en étudiant les produits scalaires entre individus :

$$G = \frac{1}{m} R^{(c)}(R^{(c)})^T$$

Les coefficients de G sont donnés par :

$$G_{ij} = \frac{1}{m}\langle x_i^{(c)}, x_j^{(c)}\rangle$$

où $x_i^{(c)}$ est la i-ème ligne centrée de R.

L'ACP classique repose donc uniquement sur les produits scalaires entre données centrées dans $\mathbb{R}^n$.

Généralisation aux noyaux

L'ACP à noyaux (*Kernel PCA*) généralise cette idée en remplaçant le produit scalaire euclidien par un produit scalaire dans un espace de Hilbert $\mathcal{H}$.
Nous considérons une application non linéaire $\phi : \mathbb{R}^n \to \mathcal{H}$, sans la calculer explicitement : seules les valeurs des produits scalaires $\langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$ sont nécessaires.
Ces produits scalaires sont évalués par une fonction noyau $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ vérifiant :

$$k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle_{\mathcal{H}}$$

Nous construisons alors la **matrice de Gram** $K \in \mathbb{M}_{m,m}(\mathbb{R})$ définie par :

$$K_{ij} = k(x_i, x_j)$$

Centrage de la matrice de Gram

Comme en ACP classique, il est nécessaire de centrer les données dans l'espace de Hilbert.
Le centrage de la matrice de Gram K s'effectue par la formule suivante :

$$\widetilde{K} = K - \frac{1}{m}\mathbf{1}_m K - \frac{1}{m} K \mathbf{1}_m + \frac{1}{m^2}\mathbf{1}_m K \mathbf{1}_m$$

où $\mathbf{1}_m$ est une matrice $m \times m$ contenant seulement des 1.
La diagonalisation de $\widetilde{K}$ permet alors d'obtenir les composantes principales dans l'espace $\mathcal{H}$, sans jamais avoir besoin de calculer explicitement les $\phi(x_i)$.

Remarque. Lorsque le noyau choisi est simplement $k(x, y) = \langle x, y \rangle$, l'ACP à noyaux se réduit exactement à l'ACP classique.

Implémentation pratique

Exemple Python

Voici un exemple simple d'utilisation de L'ACP à noyaux en Python à l'aide de la bibliothèque `scikit-learn`. Nous générons un jeu de données présentant une structure non linéaire (des cercles concentriques) et utilisons un noyau gaussien '`rbf`' pour effectuer la projection dans un espace de plus grande dimension où la séparation devient possible.

Listing 6 – ACP à noyau avec scikit-learn

```
from sklearn.decomposition import KernelPCA
from sklearn.datasets import make_circles

# Generation de donnees non-lineaires
X, y = make_circles(n_samples=400, noise=0.05, factor=0.3)
# ACP a noyau gaussien
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=10)
X_kpca = kpca.fit_transform(X)
```

Choix des noyaux

Le choix du noyau est une étape essentielle pour l'ACP à noyaux. Il définit la manière dont les distances sont mesurées dans l'espace transformé. Chaque noyau va induire une géométrie différente sur les données et peut permettre de mieux s'adapter à certaines structures par exemple.
Les principales fonctions noyaux utilisées en pratique sont résumées dans le tableau suivant :

Noyau	Expression	Paramètres
Gaussien	$\exp(-\gamma\|x-y\|^2)$	$\gamma > 0$
Polynomial	$(x^T y + c)^d$	$d \in \mathbb{N}, c \geq 0$
Sigmoïde	$\tanh(\gamma x^T y + c)$	γ, c

TABLE 2 – Principales fonctions noyau utilisées en pratique

Paramétrisation du noyau gaussien

Influence du paramètre γ

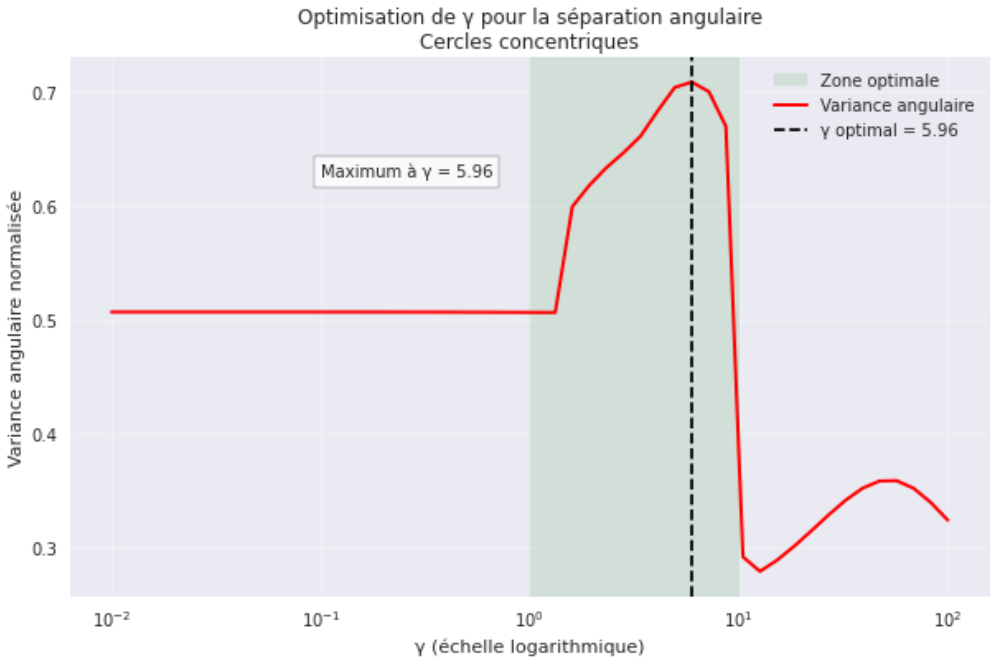


FIGURE 13 – Effet du paramètre γ sur la séparation des données.

La courbe présente trois régions caractéristiques :

1. Région sous-lissée

- **Valeur de la variance moyenne** : distribution uniforme des points
- **Comportement** :
  - Noyau RBF trop large ($k(x, y) \approx 1$)
  - Projection identique à une ACP linéaire
  - Impossible de distinguer les cercles concentriques

2. Zone optimale

- **Variance élevée**
- **Transformation géométrique** :
  - Les cercles sont "dépliés" en arcs séparés
  - Structure devenue linéarisable

3. Région de sur-ajustement

- **Chute de la variance**
- **Problèmes** :
  - Matrice de Gram quasi-diagonale ($K_{ii} \approx 1$, $K_{ij} \approx 0$)
  - Temps de calcul multiplié par 10
  - Projection dégénérée (points dispersés aléatoirement)

La valeur optimale (ici à environ 6 par validation croisée) maximise la variance entre classes tout en évitant le sur-ajustement.

Études de cas

Cercles concentriques

Un exemple classique de non-séparabilité linéaire de données est le cas des cercles concentriques. Deux classes de points sont situées à la surface de deux cercles concentriques centrés à l'origine. Ce cas démontre une limitation de l'ACP standard, qui n'est pas une opération sur les données indépendantes. L'ACP à noyaux (Kernel PCA) apporte la solution en projetant implicitement les données dans un espace de plus haute dimension où les cercles deviennent linéairement séparables. Nous étudierons comment le choix du noyau et de son paramètre γ qui permet de "déplier" cette structure.

Listing 7 – Optimisation de γ

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA, KernelPCA

#Generation des donnees
...

# Calcul de la variance angulaire
gammas = np.logspace(-2, 2, 50)
var_ang = []

for gamma in gammas:
    # Projection KernelPCA
    kpca = KernelPCA(n_components=2, kernel='rbf', gamma=gamma)
    X_proj = kpca.fit_transform(X)

    # Variance angulaire (metrique adaptee aux cercles)
    theta = np.arctan2(X_proj[:, 1], X_proj[:, 0])  # Conversion en angles
    var_ang.append(np.var(np.cos(theta)))  # Variance du cosinus des angles

# Determination de gamma optimal
gamma_opt = gammas[np.argmax(var_ang)]

# Visualisation
plt.figure(figsize=(10, 6))
...
```

- L'ACP classique donne une variance expliquée de 50% sur chaque axe
- L'ACP à noyau gaussien ($\gamma = 3$) permet une séparation nette.

Ici, la valeur $\gamma = 3$ permet un meilleur compromis entre métrique angulaire et temps de calcul tout en restant dans la zone "optimisée" vue plus tôt (figure 1) et est assez robuste pour résister au bruit.

Spirale logarithmique

Dans le cas d'une spirale, les points sont distribués suivant une structure très courbée. L'ACP linéaire projette les données de manière incohérente, alors que l'ACP à noyaux permet une séparation bien plus claire entre les bras de la spirale avec la projection dans un espace de Hilbert adapté.

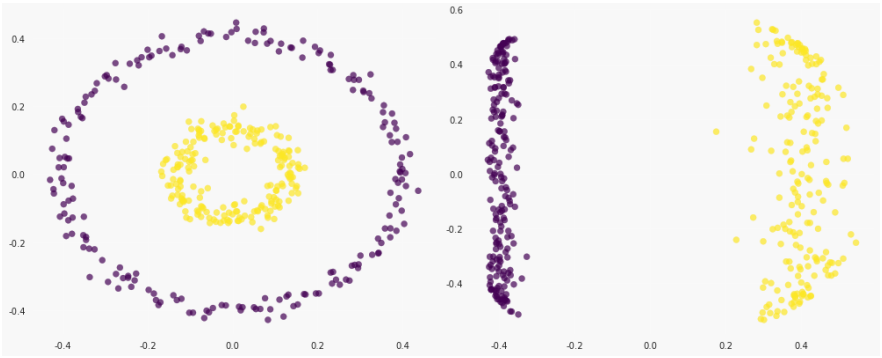
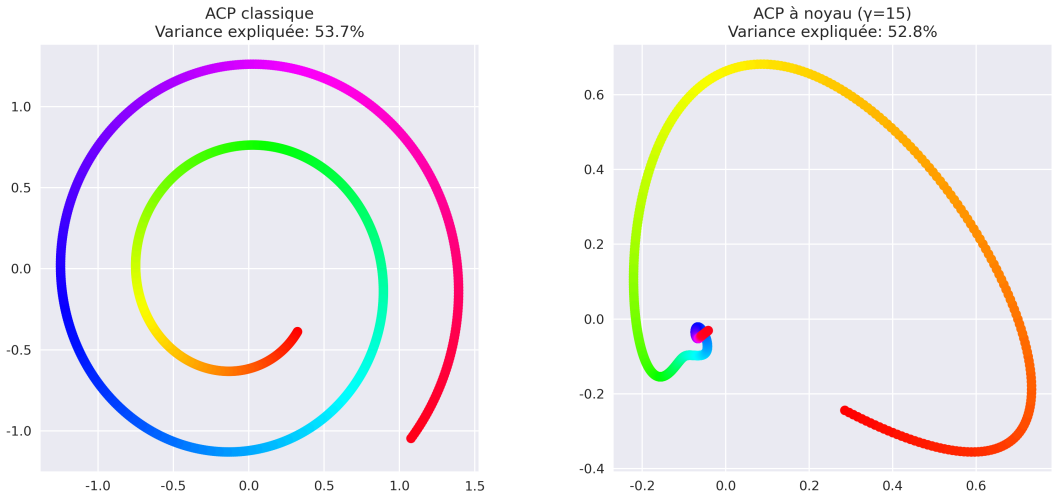


FIGURE 14 – Résultats de l'ACP classique (gauche) et ACP à noyau (droite)



(a) ACP linéaire (b) ACP noyau (RBF)

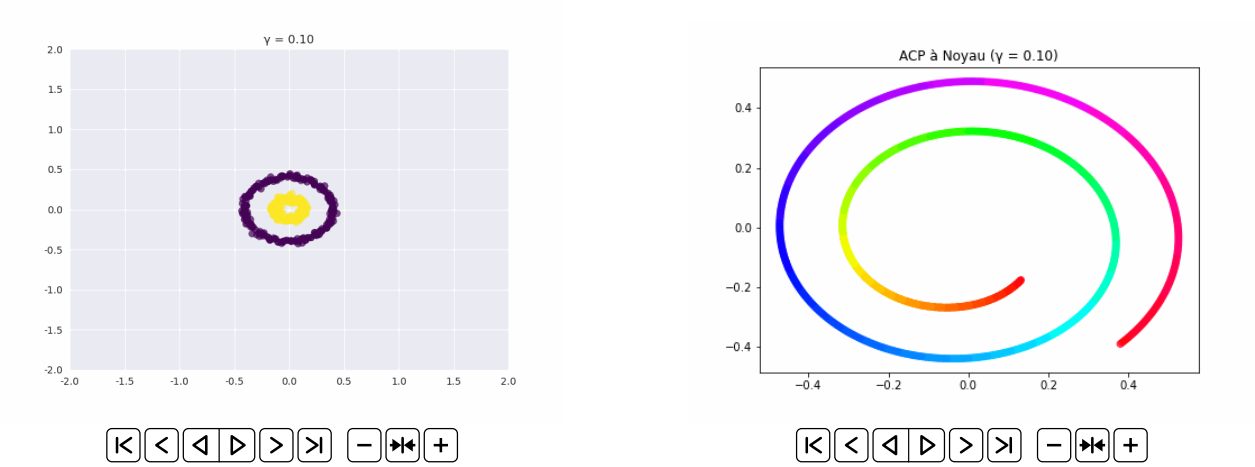
FIGURE 15 – Comparaison sur une structure spirale

Animation des projections

L'évolution des projections lors de la variation de γ est illustrée par les animations :
Nous pouvons noter le dépliage optimal autour de $\gamma \approx 3$ pour les cercles (gauche) et la linéarisation progressive de la spirale (droite).

Listing 8 – Code de génération de l'animation spirale

```
from sklearn.decomposition import KernelPCA
import matplotlib.animation as animation

fig, ax = plt.subplots()
gamma_range = np.logspace(-1, 2, 30) # echelle logarithmique

def update(gamma):
    kpca = KernelPCA(kernel='rbf', gamma=gamma)
    X_proj = kpca.fit_transform(X)
    ax.clear()
    ax.scatter(X_proj[:,0], X_proj[:,1], c=y, cmap='viridis')
    ax.set_title(f'$\gamma$ = {gamma:.1f}')

ani = animation.FuncAnimation(fig, update, frames=gamma_range, interval=300)
```



(a) Effet de γ sur les cercles concentriques

(b) Effet de γ sur la spirale

FIGURE 16 – Comparaison de l'impact du paramètre γ dans l'ACP à noyau.

```
15 ani.save('kpca_animation.gif', writer='pillow')
```

Interprétation : L'animation montre clairement comment :

- Pour $\gamma < 1$, la projection ressemble à l'ACP linéaire
- La structure optimale se voit autour de $\gamma \in [3, 15]$
- Au-delà de $\gamma = 50$, les points se dispersent complètement

Limites de cette méthode

Complexité du calcul

Ce modèle nécessite de calculer et de stocker la matrice de Gram K de taille $n \times n$, ce qui engendre une complexité de l'ordre de $O(n^3)$ pour la diagonalisation. Cette étape devient rapidement un vrai obstacle lorsque le nombre de points dépasse 10^4.

Sélection des hyperparamètres

Le choix du noyau et de ses paramètres influence fortement les résultats mais il n'y a pas de méthode universelle pour choisir le γ optimal il y donc un risque de sur-ajustement avec un γ trop grand.

Interprétabilité

Contrairement à l'ACP classique, les axes principaux n'ont plus d'interprétation directe et cela devient plus difficile d'expliquer les résultats à des non-initiés.

Chapitre 5 : Machines à Vecteurs de Supports (SVM) à noyaux

Les machines à vecteurs de support (SVM) sont des algorithmes de classification binaire qui cherchent à séparer deux classes de données par un hyperplan, tout en maximisant la marge entre les classes. Cependant, lorsque les données ne sont pas linéairement séparables, une généralisation est nécessaire grâce à la théorie des noyaux.

SVM classiques

Formulation du problème primaire

On considère un ensemble d'apprentissage :

$$\{(x_i, y_i)\}_{i=1}^m, \quad x_i \in \mathbb{R}^n, \quad y_i \in \{-1, 1\}$$

L'objectif est de trouver un hyperplan défini par :

$$f(x) = w^T x + b$$

tel que :

$$y_i(w^T x_i + b) \geq 1 \quad \text{pour tout } i$$

On souhaite maximiser la marge, ce qui équivaut à résoudre :

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{sous les contraintes} \quad y_i(w^T x_i + b) \geq 1$$

Si les données sont linéairement séparables, la SVM peut être appliquée sans problème. Voici un exemple de problèmes générés par la loi uniforme. On remarque qu'en ajustant les paramètres de la SVM, on obtient des résultats concluants.

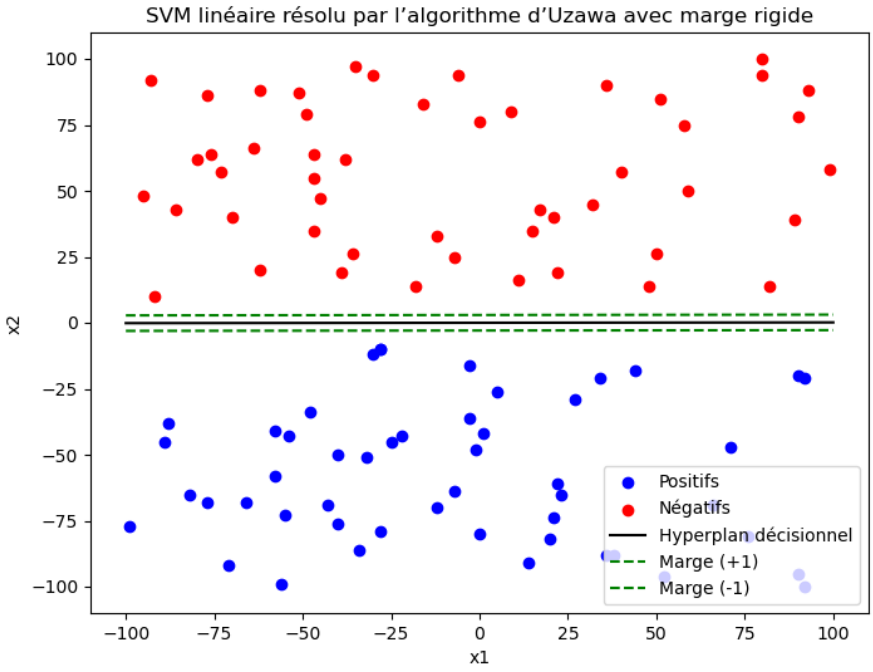


FIGURE 17 – Caption

Listing 9 – Code de la SVM classique

```python
def SVM_uzawa(u, v, gamma, gamma_b, max_iter, tol ,  p , q):
    # Initialisation
    lambda_vals = np.zeros(p)  # multiplicateurs pour les contraintes positives
    mu_vals = np.zeros(q)  # multiplicateurs pour les contraintes negatives
    b_tilde = 0.0  # biais initial
    w_tilde = np.zeros(n)  # initialisation de w_tilde

    L = [2 * tol, np.sum(lambda_vals) + np.sum(mu_vals)]
    k = 0
    # Algorithme d Uzawa
    while np.linalg.norm(L[0] - L[1]) > tol and k < max_iter:
        k += 1
        # Mise a jour des variables primales a partir des multiplicateurs actuels
        w_tilde = np.sum(lambda_vals.reshape(-1, 1) * u, axis=0) - np.sum(mu_vals.reshape(-1, 1) * v, axis=
        # Mise a jour de b_tilde par descente sur la contrainte stationnaire : sum(lambda) = sum(mu)
        b_tilde = b_tilde - gamma_b * (np.sum(lambda_vals) - np.sum(mu_vals))

        # Mise a jour des multiplicateurs par gradient ascendant (avec projection sur R+)
        for i in range(p):
            delta_lambda = 1 - u[[i], :] @ w_tilde + b_tilde
            lambda_vals[i] = np.maximum(np.zeros(1), lambda_vals[i] + gamma * delta_lambda)[0]
        for j in range(q):
            delta_mu = 1 + v[[j], :] @ w_tilde - b_tilde
            mu_vals[j] = np.maximum(np.zeros(1), mu_vals[j] + gamma * delta_mu)[0]
        L[0], L[1] = L[1], np.sum(lambda_vals) + np.sum(mu_vals)
        print(np.linalg.norm(L[0] - L[1]))

    print("w_tilde =", w_tilde)
    print("b_tilde =", b_tilde)
    print("lambda =", lambda_vals)
    print("mu =", mu_vals)

    if n == 2:
        # Generation de la ligne x pour les hyperplans
        x_line = np.linspace(-100, 100, 200)

        # Hyperplan decisionnel
        y_decision = (b_tilde - w_tilde[0] * x_line) / w_tilde[1]

        # Hyperplan de marge positive
        y_margin_pos = (b_tilde + 1 - w_tilde[0] * x_line) / w_tilde[1]

        # Hyperplan de marge negative
        y_margin_neg = (b_tilde - 1 - w_tilde[0] * x_line) / w_tilde[1]

        # Creation de la figure
        plt.figure(figsize=(8, 6))

        # Affichage des points positifs et negatifs
        plt.scatter(u[:, 0], u[:, 1], color='blue', label='Positifs')
        plt.scatter(v[:, 0], v[:, 1], color='red', label='Negatifs')

        # Trace des hyperplans
        plt.plot(x_line, y_decision, 'k-', label="Hyperplan decisionnel")
        plt.plot(x_line, y_margin_pos, 'g--', label="Marge (+1)")
        plt.plot(x_line, y_margin_neg, 'g--', label="Marge (-1)")

        # Ajustements et legendes
        plt.xlabel("x1")
        plt.ylabel("x2")
        plt.title("SVM lineaire resolu par l algorithme d Uzawa avec marge rigide")
        plt.legend()
        plt.show()
    return w_tilde, b_tilde, lambda_vals, mu_vals
```

Limitation des SVM classiques

Les SVM classiques échouent lorsque les données ne sont pas séparables de manière linéaire. De plus, il faut des classes claires qui ne se chevauchent pas et éviter les individus spéciaux susceptibles de perturber le fonctionnement de la SVM. Voici un jeu de données non séparable linéairement.

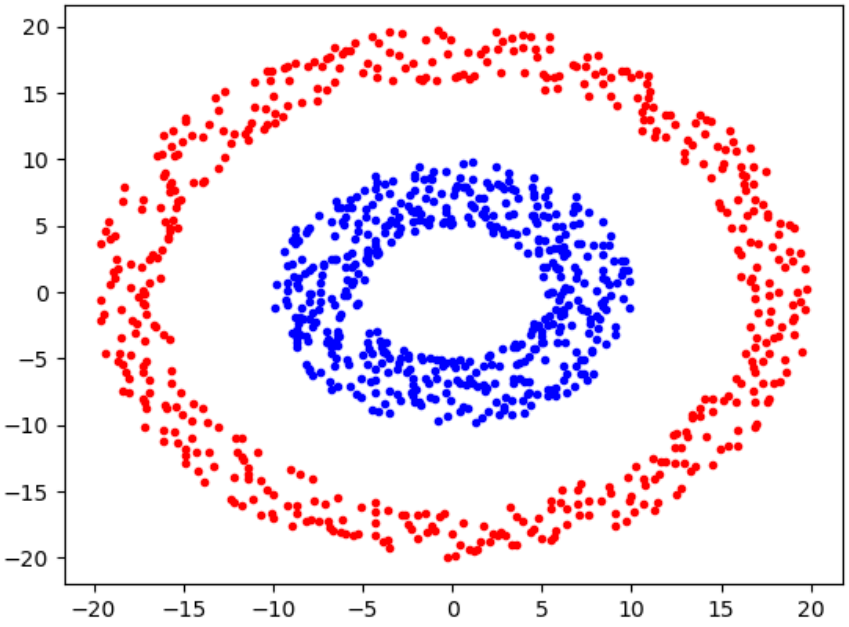


FIGURE 18 – Exemple où les classes ne sont pas séparables par un hyperplan

Généralisation grâce aux noyaux

Pour traiter des cas non linéaires, on applique une fonction $\varphi : \mathbb{R}^n \to \mathcal{H}$, qui projette les données dans un espace de Hilbert de grande dimension où elles deviennent linéairement séparables.

Cependant, calculer explicitement $\varphi(x)$ est souvent impossible. Le **trick du noyau** permet de contourner ce problème : on utilise une fonction $k(x, z)$ telle que :

$$k(x, z) = \langle \varphi(x), \varphi(z) \rangle$$

Ainsi, sans jamais calculer φ explicitement, on travaille dans l'espace transformé.

Formulation du SVM à noyaux

Le problème dual devient :

$$\max_{\alpha} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

La fonction de décision devient :

$$f(x) = \sum_{i=1}^{m} \alpha_i y_i k(x_i, x) + b$$

Exemples de noyaux

— **Noyau polynomial :**

$$k(x, z) = (\langle x, z\rangle + c)^d$$

— **Noyau gaussien (RBF) :**

$$k(x, z) = \exp\left(-\gamma\|x - z\|^2\right)$$

— **Noyau sigmoïde :**

$$k(x, z) = \tanh(\beta\langle x, z\rangle + \theta)$$

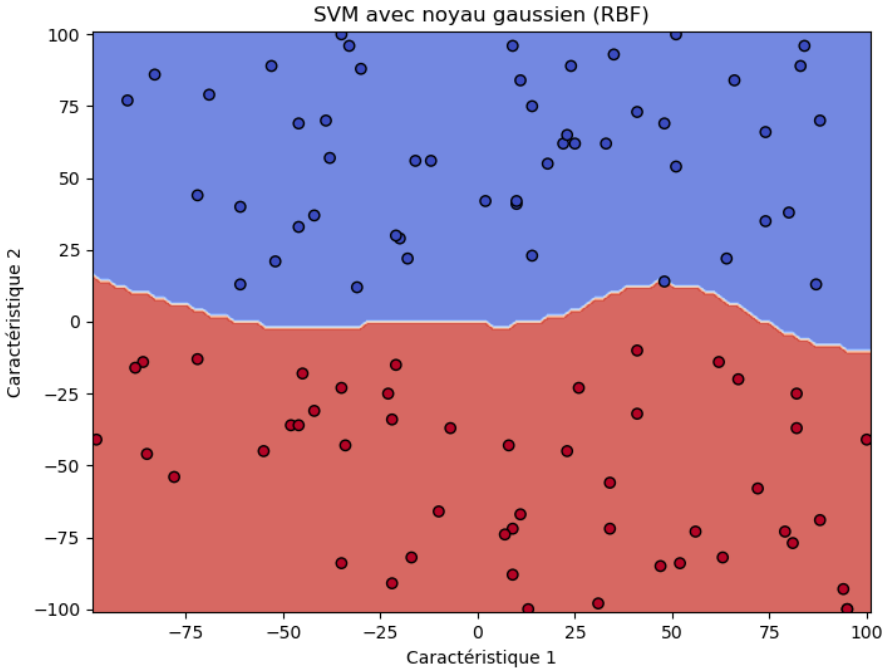


FIGURE 19 – SVM à noyaux : séparation non linéaire dans l'espace d'origine

Ici, on applique la SVM sur le même problème que précédemment, et on constate que les résultats sont similaires sur les resultat mais pas sur la maniere. En Effet on voit que . D'ailleurs, on précise qu'on utilise l'algorithme d'Uzawa pour la partie sans noyau, et qu'on a recours à la SVM avec un noyau gaussien pour la partie avec noyau. Il est important de préciser qu'une bonne sélection des paramètres est nécessaire pour obtenir des résultats intéressants. Nous les trouverons par tâtonnement.

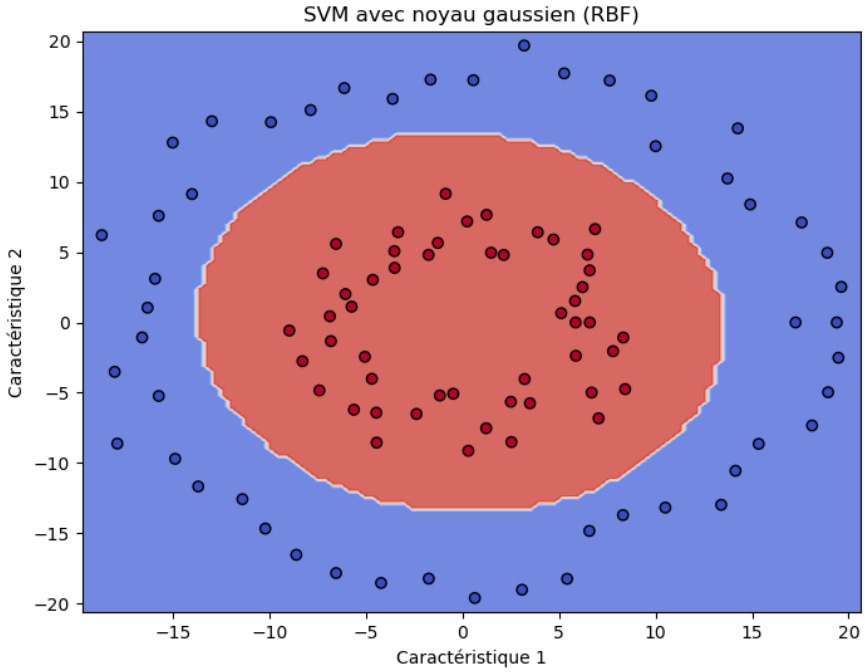


FIGURE 20 – SVM avec noyau gaussien

Listing 10 – Code de la realistion de la SVM avec Noyau Gaussien

```
1
2  import numpy as np
3  import pandas as pd
4  import matplotlib.pyplot as plt
5  import random as r
6  from sklearn import svm
7  from sklearn.datasets import make_classification
8  from sklearn.model_selection import train_test_split
9
10 def SVM_noyau_gaussien( X , y):
11     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
12
13     # Creation du modele SVM avec noyau gaussien (RBF)
14     clf = svm.SVC(kernel='rbf', C=1.0, gamma=0.001)
15
16     # Entrainement du modele
17     clf.fit(X_train, y_train)
18
19     # Evaluation du modele
20     accuracy = clf.score(X_test, y_test)
21     print(f"Precision du modele : {accuracy:.2f}")
22
23     # Visualisation des donnees et de la frontiere de decision
24     plt.figure(figsize=(8, 6))
25
26     # Creation d'une grille pour visualiser la frontiere
27     x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
28     y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
29     xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100))
30
31     # Predictions sur la grille
32     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
33     Z = Z.reshape(xx.shape)
```

```
34
35      # Affichage de la frontiere de decision
36      plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.coolwarm)
37
38      # Affichage des points de donnees
39      plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.coolwarm)
40      plt.title("SVM avec noyau gaussien (RBF)")
41      plt.xlabel("Caracteristique 1")
42      plt.ylabel("Caracteristique 2")
43      plt.show()
```

Limites de la généralisation par noyaux

— **Choix du noyau** : Un mauvais choix peut entraîner de mauvaises performances.

— **Coût computationnel** : La matrice de Gram est de taille $m \times m$.

— **Risque de sur-apprentissage** : Si le noyau est trop flexible.

La théorie des noyaux permet aux SVM de s'adapter à des problèmes complexes en projetant les données dans des espaces de grande dimension sans coût direct de calcul. C'est une méthode puissante mais nécessitant une bonne compréhension des choix de noyaux et de la complexité computationnelle.

Problème Multi-Classe

Nous n'avons travaillé que sur des données à deux dimensions, mais que se passerait-il pour des jeux de données à n dimension ? Une bonne manière de régler le problème avec ce qu'on a vu serait de se servir de l'ACP. En effet, même si des méthodes plus avancées nous permettent de généraliser les SVM à des problèmes multi-classe, il est préférable d'utiliser l'ACP. Dans ce cours, nous pouvons utiliser tout ce que nous avons développé avec l'ACP. Voici un petit exemple sur la base de données Iris.

Conclusion

Dans ce travail, nous avons étudié l'utilisation de l'astuce des noyaux afin d'étendre les méthodes d'apprentissage linéaire aux problèmes non linéaires.

En exploitant les propriétés des noyaux positifs, nous avons pu reformuler des techniques classiques telles que les moindres carrés et les machines à vecteurs de support (SVM) dans des espaces de Hilbert de dimension potentiellement infinie, avec un coût computationnel raisonnable. Cette approche permet notamment de traiter efficacement des données complexes, comme nous l'avons démontré à l'aide de plusieurs exemples (lois uniformes, normales, base Iris, Fashion-MNIST).

Toutefois, nous avons souligné que cette généralisation repose sur des choix de noyaux et de paramètres qui influencent fortement les performances du modèle. De plus, l'augmentation de la complexité computationnelle demeure un enjeu dans le traitement de grands ensembles de données.

Ainsi, l'astuce des noyaux s'affirme comme une méthode puissante, essentielle pour la modernisation des techniques d'analyse et d'apprentissage automatique classiques. Si la complexité des noyaux permet d'améliorer les performances des modèles, elle soulève également des questions d'interprétabilité. Une piste de réflexion future pourrait être le développement de méthodes permettant d'expliquer de manière plus transparente les décisions prises dans ces espaces de grande dimension.